МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«ВВЕДЕНИЕ В АРХИТЕКТУРУ x86/x86-64»

студента 2 курса, 23202 группы

Пятанова Михаила Юрьевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
Владислав Александрович
Перепёлкин

Новосибирск 2024

СОДЕРЖАНИЕ

ЦЕЛЬ

1. Знакомство с программной архитектурой x86/x86-64.
2. Анализ ассемблерного листинга программы для архитектуры x86/x86-64.

ОПИСАНИЕ РАБОТЫ

Полный компилируемый листинг реализованной программы и команда для её компиляции представлены в приложении 1.

Листинги на ассемблере с описаниями назначения команд с точки зрения реализации алгоритма выбранного варианта представлены в приложении 2 (уровень оптимизации O0) и 3 (уровень оптимизации O3).

Была определена разница в ассемблерных листингах двух уровней оптимизации. См. приложение 3.

ЗАКЛЮЧЕНИЕ

Было проведено знакомство с программной архитектурой x86/x86-64. Проанализирован ассемблерный листинг программы для архитектуры x86/x86-64.

Найдены отличия в ассемблерных листингах оптимизации уровней O0 и O3. Так в O3 отсутствую вызовы функций (кроме ввода, вывода) – все функции были встроены (инлайнинг) в код; программа работает преимущественно с регистрами, а не со стеком, ведь они быстрее. Далее программа в целом делает меньше обращений в память и меньше сравнений. Стоит отметить, что код был сильно разделен разными лейблами.

С субъективной точки зрения ассемблер после оптимизации стал читаться в разы хуже, однако его понимание оказалось крайне полезным: получилось явно увидеть оптимизационные преобразования программы компилятором.

Приложение 1.

Команды для компиляции:
gcc -S main.c -o list_O0.s -O0
gcc -S main.c -o list_O2.s -O3

```
#include <iostream>

long double GetSign(unsigned long long n) {
   return n % 2 == 0 ? 1.0 : -1.0;
}

long double GetPiNumberWithNAccuracy(unsigned long long N) {
   long double res = 0;
   for (unsigned long long i = 0; i <= N; i++) {
      res += (GetSign(i) / (2*i + 1));
   }
   return 4 * res;
}

int main() {
   unsigned long long n;
   std::cin >> n;
   std::cout << GetPiNumberWithNAccuracy(n) << "\n";
   return 0;
}
```

Приложение 2.

Ассемблерный листинг для архитектуры x86-64 с оптимизацией O0

.file "lab2.cpp"	// название файла
.text	// .globl делает iostream глобально видимой, т.е. все файлы имеют доступ
#APP	// Аналогично
.globl _ZSt21ios_base_library_initv	// GetSign() определяется как функция
#NO_APP	// определяем функцию GetSign()
.globl _Z7GetSigny	// лейбл LFB показывает, что началось тело функции (local function begin)
.type _Z7GetSigny, @function	// Call Frame Information. Директива используется для отладки и обработки искл.
_Z7GetSigny:	// %rbp — это указатель базы, который указывает на базу текущего стекового фрейма.
.LFB1988:	// CFA используется для освобождения стека во время обработки исключений или отладки.
.cfi_startproc	// %rsp — указатель стека, который указывает на вершину текущего стекового фрейма. Копируем указатель фрейма, записываем в регистр rbp значение регистра rsp - это будет начало фрейма для локальных переменных.
endbr64	// копируем входное значение (занимающее 8 байт) функции с регистра %rdi (на нем в первую очередь передается) в стек
pushq %rbp	// копируем 8-байтное значение в %rax регистр арифметических операций
.cfi_def_cfa_offset 16	// побитовый оператор «и» между 1 и регистра %eax (32 битная часть %rax) по сути изолирует младший бит
.cfi_offset 6, -16	// логическое сравнение %rax и %rax, а далее ветвление: если 1, то L2, иначе (1) помещаем одну из семи часто используемых констант (в формате с плавающей запятой двойной расширенной точности) в стек регистров FPU и запускаем L4
movq %rsp, %rbp	// L2 добавляем 1 в стек и сменяем его знак. (т.е., по сути, -1.0 в стеке)
.cfi_def_cfa_register 6	// L4 удаляем значение со стека, копируя его в rbp
movq %rdi, -8(%rbp)	// ret return возвращает значение со стека
movq -8(%rbp), %rax	Таким хитрым образом, побитовой конъюнкцией, была реализована проверка на четность числа
andl $1, %eax	
testq %rax, %rax	
jne .L2	
fld1	
jmp .L4	
.L2:	
fld1	
fchs	
.L4:	
popq %rbp	
.cfi_def_cfa 7, 8	
ret	
.cfi_endproc	
.LFE1988:	
.size _Z7GetSigny, .-_Z7GetSigny	
.globl _Z24GetPiNumberWithNAccuracyy	
.type _Z24GetPiNumberWithNAccuracyy, @function	// Аналогично инициализируем функцию GetPiNumberWithNAccuracy()
_Z24GetPiNumberWithNAccuracyy:	
.LFB1989:	
.cfi_startproc	
endbr64	// %rsp — указатель стека, который указывает на вершину текущего стекового фрейма. Копируем указатель фрейма, записываем в регистр rbp значение регистра rsp - это будет начало фрейма для локальных переменных.
pushq %rbp	
.cfi_def_cfa_offset 16	
.cfi_offset 6, -16	
movq %rsp, %rbp	
.cfi_def_cfa_register 6	// резервируем 48 байт на стеке вычитанием
subq $48, %rsp	Копируем в место на стеке (на -40 месте) значение из %rdi (аргумент функции)
movq %rdi, -40(%rbp)	Загружаем на стек +0.0
fldz	Команда fstpt копирует значение из регистра ST(0) в ячейку памяти (long double res = 0)
fstpt -16(%rbp)	Копируем в место на стеке (на -24 месте) 0 (i = 0)
movq $0, -24(%rbp)	Переходим в L6
jmp .L6	Копируем на регистр %rax со стека (i)
.L8:	Передаем значение с %rax в %rdi и вызываем функию (как раз благодаря %rdi мы передаеем в неё значение) GetSign(i)
movq -24(%rbp), %rax	Вновь копируем на регистр %rax со стека (i)
movq %rax, %rdi	Прибавляем тоже самое значение (по сути умножение на 2)
call _Z7GetSigny	Прибавляем 1 (2*i + 1)
movq -24(%rbp), %rax	Результат записываем на стек в -48
addq %rax, %rax	Преобразует исходный операнд из целого числа со знаком в формат с плавающей запятой двойной расширенной точности и помещает значение в стек регистров FPU в st(0), а res сдвигается в st(1) По сути преобр. (2*i + 1) в double
addq $1, %rax	Сравнивает значения в %rax
movq %rax, -48(%rbp)	jns: выполняет переход к метке, если флаг знака не установлен (т.е. число неотрицательное) Это условие всегда выполняется в силу условий задачи
fildq -48(%rbp)	кладет в стек FPU значение с .LC4 (очень маленькое число?)
testq %rax, %rax	Складывает st(0) с st(1) и записывает в st(1) делает pop
jns .L7	
fldt .LC4(%rip)	Делит st(1) на st(0) и записывает в st(1) и делает pop
faddp %st, %st(1)	Добавляет на стек FPU значение на -16 месте со стека
.L7:	Складывает st(0) с st(1) и записывает в st(1) делает pop
fdivrp %st, %st(1)	Команда fstpt копирует значение из регистра ST(0) в ячейку памяти (res += …
fldt -16(%rbp)	Прибавляем 1 к стеку (i++)
faddp %st, %st(1)	
fstpt -16(%rbp)	Копируем со стека в арифметический регистр %rax
addq $1, -24(%rbp)	И сравниваем с другой переменной, хранящийся на -40 позиции
.L6:	Если меньше или равно прыгаем в L8 (i <= N)
movq -24(%rbp), %rax	
cmpq %rax, -40(%rbp)	
jnb .L8	

fldt -16(%rbp)	Пушим на FPU стек res со стека
fldt .LC5(%rip)	Пушим на FPU стек .LC5(4)
fmulp %st, %st(1)	Умножаем ST(0) * ST(1) = 4 * res и pop
leave	Освобождаем стек. Так мы восстанавливаем состояние стека и кадра, которые были до вызова
.cfi_def_cfa 7, 8	Покидаем функцию
ret	
.cfi_endproc	
.LFE1989:	
.size _Z24GetPiNumberWithNAccuracyy, .-_Z24GetPiNumberWithNAccuracyy	
.section .rodata	
.LC6:	
.string "\n"	
.text	
.globl main	
.type main, @function	
main:	
.LFB1990:	Определяем функцию main
.cfi_startproc	
endbr64	
pushq %rbp	%rsp — указатель стека, который указывает на вершину текущего стекового фрейма. Копируем указатель фрейма, записываем в регистр rbp значение регистра rsp — это будет начало фрейма для локальных переменных
.cfi_def_cfa_offset 16	
.cfi_offset 6, -16	
movq %rsp, %rbp	
.cfi_def_cfa_register 6	
subq $16, %rsp	Резервируем 16 байт на стеке
movq %fs:40, %rax	Защита стека
movq %rax, -8(%rbp)	Записываем значение с %rax на стек
xorl %eax, %eax	Обнуляем %eax
leaq -16(%rbp), %rax	Записываем эффективный адрес из стека в %rax
movq %rax, %rsi	Записываем %rax в %rdi
leaq _ZSt3cin(%rip), %rax	Записываем значение с std::cin в %rax
movq %rax, %rdi	%rax записываем в %rdi
call _ZNSirsERy@PLT	Вызывается функция std::istream::operator>> для считывания входных данных в локальную переменную. (std::cin >> n;)
movq -16(%rbp), %rax	
movq %rax, %rdi	Передаем %rax в %rdi и вызываем функцию GetPiNumberWithNAccuracy()
call _Z24GetPiNumberWithNAccuracyy	
leaq -16(%rsp), %rsp	Кладем в %rsp со стека
fstpt (%rsp)	Копируем со стека сопроцессора в %rsp
leaq _ZSt4cout(%rip), %rax	Записываем эффективный адрес std::cout в %rax
movq %rax, %rdi	Передаем %rax в %rdi
call _ZNSolsEe@PLT	Вызываем std::ostream::operator<< чтобы вывести double
addq $16, %rsp	Возвращаем указатель на верхушку стека в начальную позицию
movq %rax, %rdx	Записываем %rax в %rdx
leaq .LC6(%rip), %rax	Записываем эффективный адрес строки в %rax (‘\n’)
movq %rax, %rsi	Записываем %rax в %rsi (второй аргумент функции)
movq %rdx, %rdi	Записываем %rdx в %rdi (первый аргумент функции)
call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT	Вызов std::ostream::operator<< чтобы вывести строку
movl $0, %eax	Устанавливаем возвращаемое значение 0
movq -8(%rbp), %rdx	Записываем в значение со стека в %rdx
subq %fs:40, %rdx	Защита стека
je .L12	Защита стека
call __stack_chk_fail@PLT	Если со стеком всё хорошо то L12
.L12:	
leave	Освобождаем стек. Так мы восстанавливаем состояние стека и кадра, которые были до вызова
.cfi_def_cfa 7, 8	Выход из функции
ret	
.cfi_endproc	
.LFE1990:	
.size main, .-main	В этой части определяются некоторые доступные только для чтения объекты данных, связанные с преобразованием целых чисел в символьные, что указывает на то, что некоторые целочисленные типы являются беззнаковыми
.section .rodata	
.type _ZNSt8__detail30__integer_to_chars_is_unsignedIjEE, @object	
.size _ZNSt8__detail30__integer_to_chars_is_unsignedIjEE, 1	
_ZNSt8__detail30__integer_to_chars_is_unsignedIjEE:	
.byte 1	
.type _ZNSt8__detail30__integer_to_chars_is_unsignedImEE, @object	
.size _ZNSt8__detail30__integer_to_chars_is_unsignedImEE, 1	
_ZNSt8__detail30__integer_to_chars_is_unsignedImEE:	
.byte 1	
.type _ZNSt8__detail30__integer_to_chars_is_unsignedIyEE, @object	
.size _ZNSt8__detail30__integer_to_chars_is_unsignedIyEE, 1	
_ZNSt8__detail30__integer_to_chars_is_unsignedIyEE:	
.byte 1	
.align 16	
.LC4:	

```
    .long   0
    .long   -2147483648
    .long   16447
    .long   0
    .align 16
.LC5:
    .long   0
    .long   -2147483648
    .long   16385
    .long   0
    .ident  "GCC: (Ubuntu 13.2.0-23ubuntu4) 13.2.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
    .long   4f - 1f
    .long   5
0:
    .string "GNU"
1:
    .align 8
    .long   0xc0000002
    .long   3f - 2f
2:
    .long   0x3
3:
    .align 8
4:
```

Приложение 3.

Ассемблерный листинг для архитектуры x86-64 с оптимизацией O3

.file "lab2.cpp"	
.text	
#APP	
.globl _ZSt21ios_base_library_initv	
#NO_APP	
.p2align 4	
.globl _Z7GetSigny	
.type _Z7GetSigny, @function	
_Z7GetSigny:	
.LFB2057:	GetSign() определяется как функция
.cfi_startproc	
endbr64	
andl $1, %edi	Побитовое и с аргументом функции (n), по сути, изолируем младший бит (который отвечает за четность числа)
fld1	Загружаем в стек сопроцессора 1
jne .L3	L3 если ZF(zero flag) = не установлен, т.е. если последняя логическая или арифметическая операция не дала 0.
ret	Выходим из функции
.p2align 4,,10	
.p2align 3	
.L3:	Меняем знак числа (на стеке) на противоположный
fchs	Выходим. **Оптимизация состоит в меньшем количестве обращений в память и сравнений**
ret	
.cfi_endproc	
.LFE2057:	
.size _Z7GetSigny, .-_Z7GetSigny	
.p2align 4	
.globl _Z24GetPiNumberWithNAccuracyy	GetPiNumberWithNAccuracy() определяется как функция
.type _Z24GetPiNumberWithNAccuracyy, @function	
_Z24GetPiNumberWithNAccuracyy:	
.LFB2058:	
.cfi_startproc	
endbr64	Загружаем +0 на стек сопроцессора **(res = 0 далее результат будет лежать в sp(3))**
fldz	Передаем 1 в регистр %edx
movl $1, %edx	Делаем ксор (искл. или) регистрв %eax и записываем в %eax. По сути, обнуляем %eax **(eax = i = 0) (оптимизация – вместо стека стараемся использовать более быстрые регистры)**
xorl %eax, %eax	Загружаем +1 на стек сопроцессора **(оптимизация - появляются две вспомогательные переменные – значения GetSign())**
fld1	Загружаем +1 на стек сопроцессора
fld1	Меняем знак числа (st(0)) на противоположный
fchs	
.p2align 4,,10	
.p2align 3	
.L11:	Записываем %rdx в стек на -16 позицию
movq %rdx, -16(%rsp)	Пушим на стек сопроцессора конвертированное -16(%rsp) в вещ. число
fildq -16(%rsp)	Сравниваем младшие 8бит %rax с 1 (ZF = 1 если результат 0) **оптимизация:**
testb $1, %al	Если младший бит равен 0, то L6 ZF = 1 (**по сути проверяем четность - была выполнена подстановка функции GetSign(), чтобы её не вызывать**)
je .L6	Сравниваем %rdx, смотрим на знак
testq %rdx, %rdx	Если отрицательное, то L13
js .L13	
.L7:	Обратное деление(r): st(1) / st(0) и записываем в st(0) **(-1 / (2*i + 1))**
fdivr %st(1), %st	
.L12:	Складываем 1 с %rax **(i++)**
addq $1, %rax	Складываем st(0) с st(3) (и оставляем там и делаем pop) **(res += …)**
faddp %st, %st(3)	Складываем 2 с %rdx **(оптимизация как бы появился второй счетчик, бегущий по нечетным начиная с 1 (2i + 1))**
addq $2, %rdx	Сравниваем %rax с %rdi (i <= N) **опять же экономим память на стеке**
cmpq %rax, %rdi	Если %rdi больше или равно %rax,
jnb .L11	то прыгаем в L11
fstp %st(0)	Записываем значение st(0) в st(0) и pop
fstp %st(0)	Записываем значение st(0) в st(0) и pop - > **эффективно избавляемся от двух верхних значений стека сопроцессора**
fmuls .LC5(%rip)	Умножаем st(0) на 4 **return 4 * res**
ret	Выходим из функции
.p2align 4,,10	
.p2align 3	
.L13:	Складываем s(0) с .LC4
fadds .LC4(%rip)	Прыгаем в L7
jmp .L7	
.p2align 4,,10	
.p2align 3	
.L6:	Сравниваем %rdx с %rdx
testq %rdx, %rdx	Если число отрицательное, то L14
js .L14	Обратное деление(r): st(2) / st(0) и записываем в st(0) **(1 / (2*i + 1)**
fdivr %st(2), %st	Прыгаем в L12
jmp .L12	
.p2align 4,,10	
.p2align 3	

.L14:	
fadds .LC4(%rip)	Складывает s(0) .LC4
fdivr %st(2), %st	Обратное деление(r): st(2) / st(0) и записываем в st(0) **(GetSign(i) / (2*i + 1)**
jmp .L12	Прыгаем в L12
.cfi_endproc	
.LFE2058:	
.size _Z24GetPiNumberWithNAccuracyy, .-_Z24GetPiNumberWithNAccuracyy	
.section .rodata.str1.1,"aMS",@progbits,1	
.LC7:	
.string "\n"	
.section .text.startup,"ax",@progbits	
.p2align 4	
.globl main	
.type main, @function	
main:	main() определяется как функция
.LFB2059:	
.cfi_startproc	
endbr64	
subq $40, %rsp	Резервируем 40 байт на стеке
.cfi_def_cfa_offset 48	
leaq _ZSt3cin(%rip), %rdi	Записываем значение с std::cin в %rdi **(сразу в rdi минуя rax)**
movq %fs:40, %rax	Защита стека
movq %rax, 24(%rsp)	Записываем в стек на 24 месте **(работаем без rbp)**
xorl %eax, %eax	Обнуляем %rax
leaq 16(%rsp), %rsi	Записываем эффективный адрес из стека в %rsi
call _ZNSi10_M_extractIyEERSiRT_@PLT	Вызывается класс из c++ **std::string**, записываем ввод в N
movq 16(%rsp), %rcx	Записываем со стека в %rcx (N)
movl $1, %edx	Записываем 1 в %edx
xorl %eax, %eax	Обнуляем %eax
fldz	**Оптимизация: была выполнена подстановка функции**
fld1	**GetPiNumberWithNAccuracy (), чтобы её не вызывать**
fld1	**Код совпадает с описанным выше**
fchs	
.p2align 4,,10	
.p2align 3	
.L21:	
movq %rdx, 8(%rsp)	
fildq 8(%rsp)	
testb $1, %al	
je .L16	
testq %rdx, %rdx	
js .L25	
.L17:	
fdivr %st(1), %st	
.L24:	
addq $1, %rax	
faddp %st, %st(3)	
addq $2, %rdx	
cmpq %rax, %rcx	
jnb .L21	
fstp %st(0)	
fstp %st(0)	
fmuls .LC5(%rip)	Умножаем на st(0)(res)*4
subq $16, %rsp	Резервируем 16 байт
.cfi_def_cfa_offset 64	
leaq _ZSt4cout(%rip), %rdi	Записываем эффективный адрес std::cout в %rdi
fstpt (%rsp)	Сохранить вещественное значение st0(res) с извлечением из стека сопроцессора в %rsp **(вывод res)**
call _ZNSo9_M_insertIeEERSoT_@PLT	Вызываем класс **std::ostream**
leaq .LC7(%rip), %rsi	Записываем эффективный адрес .LC7 в %rsi
movq %rax, %rdi	Записываем %rax в %rdi(регистр для передачи в функцию)
popq %rax	Записывает в %rax значение со стека
.cfi_def_cfa_offset 56	
popq %rdx	Записывает в %rdx значение со стека
.cfi_def_cfa_offset 48	
movl $1, %edx	Записывает в 1 %edx
call _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l@PLT	Вызывает поток вывода
movq 24(%rsp), %rax	Записывает в %rax значение со стека с позиции 24
subq %fs:40, %rax	Вычитает 64-разрядное значение по адресу памяти %fs:40 из значения в %rax.
jne .L26	Прыгаем на L26, если значение в %rax != 0
xorl %eax, %eax	Обнуляем %eax чтобы **return 0**
addq $40, %rsp	Восстанавливаем начальный указатель на стек
.cfi_remember_state	
.cfi_def_cfa_offset 8	
ret	Выходим из функции
.p2align 4,,10	
.p2align 3	

.L25: .cfi_restore_state fadds .LC4(%rip) jmp .L17 .p2align 4,,10 .p2align 3 .L16: testq %rdx, %rdx js .L27 fdivr %st(2), %st jmp .L24 .p2align 4,,10 .p2align 3 .L27: fadds .LC4(%rip) fdivr %st(2), %st jmp .L24 .L26: call __stack_chk_fail@PLT .cfi_endproc .LFE2059: .size main, .-main .section .rodata.cst4,"aM",@progbits,4 .align 4 .LC4: .long 1602224128 .align 4 .LC5: .long 1082130432 .ident "GCC: (Ubuntu 13.2.0-23ubuntu4) 13.2.0" .section .note.GNU-stack,"",@progbits .section .note.gnu.property,"a" .align 8 .long 1f - 0f .long 4f - 1f .long 5 0: .string "GNU" 1: .align 8 .long 0xc0000002 .long 3f - 2f 2: .long 0x3 3: .align 8 4:	Ошибка на стеке, спасаем программу